Parallelization of the Laplace Smoothing Algorithm

Jeannette Spühler, Rodrigo Vilela de Abreu and Kaspar Müller
Royal Institute of Technology, Sweden

1 Introduction

This report is part of the PDC summer school 2010. We set our task to parallelize the Laplacian Smoothing algorithm in Dolfin [3].
We first introduce the algorithm, explain then the different steps of its implementation, present a performance model and finalize our report with the examination of the scalability of our code.

2 Laplacian Mesh Smoothing

Mesh smoothing plays an important role in surface reconstruction for visualization and solving finite element methods. The quality of the mesh influences the solution of a finite element problem. It has been shown in [2] that a maximum angle is essential for finding an approximate solution in two-dimensional cases and a too small angle leads to a bad condition number for the element matrix as mentioned in [4].
To enhance the quality of the mesh, one has the following options:

1. Change the number of vertices by coarsening or refining the mesh.
2. Keep the number of vertices:
   (a) Conserve the topology by relocating the vertices.
   (b) Change the topology by swapping the faces and edges.

The Laplacian Smoothing belongs to category 2a and is a very common and simple method to enhance the quality of a mesh to a reasonable level.
Its name comes from the diffusion equation $\frac{\partial u}{\partial t} - \nabla(k\nabla u) = 0$ where k is the diffusion coefficient and u is a function defining the position of a vertex in the next time step. If we approximate this Laplace equation by an equidistant grid, we see that the coordinates of a vertex are calculated as the mean value of the positions of the surrounding neighbors: $x_i = \frac{1}{N}\sum_{j=0,i\neq j}^{N} x_j$ where x_i and x_j are adjacent.

3 Algorithm

3.1 In Serial

The serial Lapacian mesh smoothing algorithm can be abstracted as follows:

1. Loop over all vertices and for each vertex:
   (a) Calculate the new vertex position from the average position of its neighbors.
   (b) Move vertex.

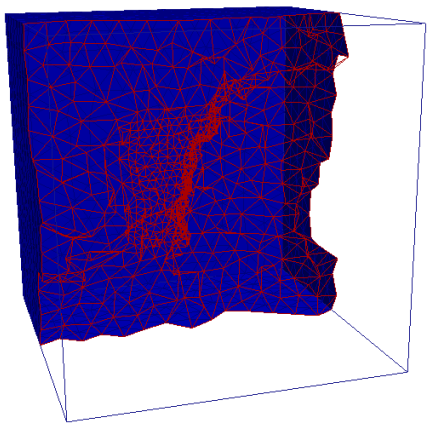

Figure 1: The global boundary is colored with blue and the interior boundary with red. Together they form the local boundary of one processor defined on a cube.

3.2 In Parallel

In a parallel setting, the movement of the nodes lying at the interface between two processors needs to be performed carefully. Since a processor has no information about the nodes located in the other processors, it has to communicate with them in order to correctly execute the Laplacian smooth algorithm. Moreover, in Dolfin-hpc, the mesh overlap between two adjacent processors is replicated on each core as ghosted entities. Each mesh entity has an assigned owner, which is responsible for keeping the data updated, [5].

We have chosen to follow this specific guideline in our parallel Laplacian smoothing, such that only the processors owning a node are allowed to move it. We present in the following part a possible implementation of the communication and computation to smooth the mesh in parallel. The algorithm implemented for this report can be divided into four steps.

Step 1: Prepare for communication and smoothing.

Step 2: Communicate data for smoothing

Step 3: Perform Smoothing

Step 4: Communicate changed node positions

While implementing the algorithm the following properties need to be considered. Nodes, seen by a processor as ghost nodes are vertices on the processors boundary owned by the neighboring processor. In Dolfin-hpc, the processor knows the neighbor who owns the ghost vertex, but the neighbor does not know which processors sees his owned vertex as a ghost node. When smoothing the mesh according to the algorithm sketched above, only the owned nodes are moved. Thus, after all nodes have been moved, their new positions need to be transmitted to the other processors having them as ghosts.

3.2.1 Step 1: Prepare for communication and smoothing

Each processor build the following three sorted containers by iterating over its interior boundary. The interior boundary of a processor is the interface between him and its neighbors (See Fig. 1).

- *owner_tree*

  Key: *rank* of ghost node owner

  Element: Global index of ghost node

- *send_inner*

  Key: Global index of ghost node

  Element1: Number of neighboring nodes

  Element2: Sum over x-coordinate of neighboring nodes

  Element3: Sum over y-coordinate of neighboring nodes

  Element4: Sum over z-coordinate of neighboring nodes

- *recv_sum*

  Key: Global index of owned node

  Element1: Number of neighboring nodes

  Element2: Sum over x-coordinate of neighboring nodes

  Element3: Sum over y-coordinate of neighboring nodes

  Element4: Sum over z-coordinate of neighboring nodes

As mentioned above, since only the owner of a node is allowed to move the position of the node, we have to send the missing data to the owner. The container *owner_tree* is used to keep track of the owner of the corresponding ghost node. The corresponding data to send is stored in *send_inner*. Later in Step 2, the message is assembled by looking up the receiver in *owner_tree* and attaching to it the corresponding data stored in *send_inner*. The third container *recv_sum* is built to gather date of the owned nodes. Later in Step 3, this data is completed with the received data from the neighboring processors.

3.2.2 Step 2: Communicate data for smoothing

In the second part of the implementation, the communication is performed. The communication between the source and destination processors is defined by a standard formula, where all processors communicate with each other, even if they have no information to exchange (in which case they exchange an empty message). It reads:

```
for j < P:

  src = (rank -j + P) % P;
  dest = (rank + j) % P;

  ...

  MPI_Sendrecv(&send_buff[0], ..., dest, recv_buff, ..., src, ...);

end
```

where P is the total number of processors running in parallel, and *rank* is the processor number. The dots before the MPI call indicating the construction of *send_buff*. It contains the global index of the ghost point of *src* as well as the number of neighbors and the sums of the x, y resp. z component. In this way *dest* receives in *recv_buff* the missing data to compute the smoothing. Unimportant variables in the MPI call where replaced by dots. While receiving data, the sender of the message is stored in a new container called *ghost_tree*.

- *ghost_tree*

  Key: *rank* of proc. who sees the owned node as a ghost node

  Element: Global index of owned node

In this way we can keep track of the processors which need to receive the updated position of the node after smoothing.

3.2.3 Step 3: Perform smoothing

To perform the smoothing each processor pass through all his owned nodes in his domain. Thereby he distinguishes between tree cases.

Case 1: Node is on global boundary

Case 2: Node is on interior boundary

Case 3: Node is not on boundary

In Case 1 the nodes are skipped, since the global boundary is steady in our implementation. In Case 2 the information collected in *recv_sum* is completed with the received data and the smoothing is performed. In Case 3 the vertices are moved with no additional consideration.

3.2.4 Step 4: Communicate changed node positions

After all processors have smoothed their mesh, the updated positions of the vertices acting as ghost nodes in other processors are reported back by using *ghost_tree*.

4 Performance model

We present next a performance model for the algorithm described above. The model is derived exclusively for the second part of the algorithm, where all data is computed, exchanged, and the actual smoothing of the mesh is carried out. We begin by identifying two distinct parts in the model, a serial part that is local to each processor, and is related to the time it takes to perform the arithmetic operations necessary to compute the new coordinates, and a second one that accounts for the communication between the processors working in parallel. We assume that the time spent in the first part is proportional to the average number of nodes per processor, N_c/P (where N_c is the total number of nodes in the mesh and P is the number of processors used), and to the average number of neighbors for each node in the mesh, named by c. Then, since we know that for each neighboring node we have three additions (which correspond to the three coordinates in a three-dimensional mesh), and one division, we may model the serial computational time, T_{loc}, by:

$$T_{loc} = O(N_c/P \cdot (3c + 1) \cdot \tau_f),$$

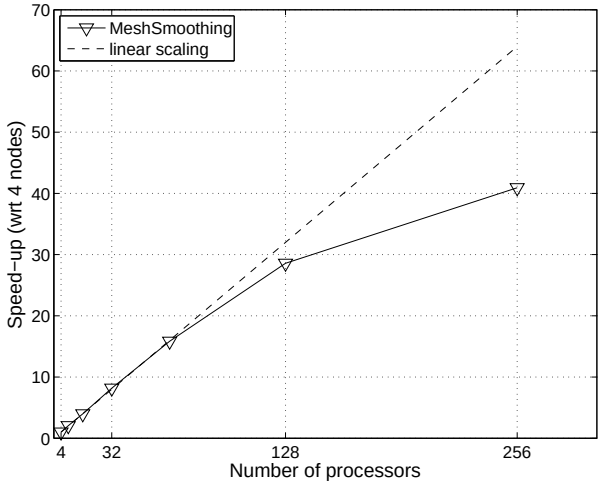

Figure 2: Strong scaling result for our implementation of the Laplacian Mesh Smoothing algorithm.

where τ_f is the time required to execute one flop. The communication time, T_{comm}, on the other hand, is assumed to be proportional to the average number of shared nodes in each processor, N_s, and, if all processors communicate with each other, we may write:

$$T_{comm} = (P-1) \cdot O(\tau_s + N_s \cdot m_{size} \cdot \tau_b),$$

where τ_s is the latency time of the interconnect, m_{size} is the average message size in $Bytes$, and τ_b is the message passing time per $Byte$, which is given by the bandwidth of the network. The total theoretical time for our smoothing algorithm is therefore:

$$T_{tot} = T_{loc} + T_{comm}.$$

The scaling results presented in the next session were obtained using the computer Lindgren at PDC, KTH, which is a Cray XE6 system, based on the AMD Opteron 12-core "Magny-Cours" (2.1 GHz) processors and the Cray Gemini interconnect technology. We have found in the literature, [1], that, for such systems, the values of τ_f, τ_s and τ_b are typically $1.085 \cdot 10^{-10} s/flop$ (given a peak performance of $9.2 Gflop/s$), $1.5 \cdot 10^{-6} s$ and $2 \cdot 10^{-10} s/Byte$ (assuming a bandwidth of $5 Gbytes/s$). We immediately see that communication is the most expensive part of the algorithm: for example, if $c = 6$, $N_c = 150,000$, $P = 4$, $N_s = 5,000$, $m_{size} = 48 Bytes$, and if we take the proportionality constants corresponding to the $O(...)$ terms equal to 1, we obtain $T_{loc} = 7.7 \cdot 10^{-5} s$ and $T_{comm} = 1.4 s$, i.e. the communication time is several orders of magnitude larger than the serial time required to compute the new mesh coordinates.

5 Scaling

We tested our parallel implementation of the Laplace smoothing algorithm on Lindgren at PDC. The strong scaling result is shown in Fig.2. We obtain a quasi-linear scaling up to 256 cores, which we consider satisfactory for a first implementation attempt. In order to understand why our implementation does not scale when $P > 256$, we need to scrutinize the theoretical model proposed above. As the number of processors increases, the latency time of the interconnect

dominates the communication cost since it scales linearly with P, whereas the actual data transfer cost decreases because the number of shared nodes N_s decreases faster than $1/P$. In this way, if we want to improve the scalability of our code, we need to change the communication pattern between the processors in order to keep the latency time small for even larger values of P.

References

[1] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *Proceedings of the 18th IEEE Symposium on High Performance Interconnects*, 2010.

[2] I. Babuska and A. K. Aziz. On the angle condition in the finite element method. *SIAM J. Numer. Anal. 13,*, pages 214–226, 1976.

[3] DOLFIN. *http://www.fenicsproject.org/wiki/DOLFIN*.

[4] Lori A. Freitag and Carl Ollivier-gooch. Tetrahedral mesh improvement using swapping and smoothing. *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*, 40(21):3979–4002, 1997.

[5] Niclas Jansson, Johan Hoffman, and Johan Jansson. Performance of Dolfin and Unicorn on Modern High-Performance Distributed Memory Architectures. Technical Report KTH-CTL-4012, Computational Technology Laboratory, 2010. http://www.publ.kth.se/trita/ctl-4/012/.